Precept 2: Non-preemptive Scheduler

COS 318: Fall 2018
Project 2 Schedule

- **Precept:** Monday 10/01, 7:30pm
  - (You are here)
- **Design Review:** Monday 10/08, 3 - 7pm
- **Due:** Sunday 10/14, 11:55pm
Project 2 Overview

- Goal: Build a non-preemptive kernel that can switch between different tasks (task = process or kernel thread)
- Read the project spec for more details
- Start early
What is a Non-Preemptive Kernel?

Current running task loses CPU or running state in the following scenarios:

1. **Yield**
2. **Block**: I/O operation, Lock (thread)
3. **Exit**
What is a Non-Preemptive Kernel?

COS 318:
go_to_class();
go_to_precept();
yield();
thinking();
design_review();
yield();
coding();
exit();

Life:

have_fun();
yield();
play();
yield();
do_random_stuff();
yield();
...
What is a Non-Preemptive Kernel?

COS 318:
- go_to_class();
- go_to_precept();
- yield();
- thinking();
- design_review();
- yield();
- coding();
- exit();

Life:
- have_fun();
- yield();
- play();
- yield();
- do_random_stuff();
- yield();
- ...

What You Need to Deal With

1. Process Control Blocks (PCBs)
2. User and Kernel Stack
3. Basic System Call Mechanism
4. Context Switching
5. Mutual Exclusion
Assumptions

- **Protected Mode:** No need to worry about segment registers again!
- **Non-Preemptive Tasks:** Run until they yield, block, or exit
- **Fixed Number of Tasks:** Allocate per-task state (PCB) statically in your program
- **Fixed Task Stack Size**
1. Process Control Block (PCB)

- Defined in `kernel.h` and initialized in `kernel.c:_start`
- What is its purpose?
- What should be in the PCB?
  - Process ID (PID)
  - Stack Info
  - Registers
  - CPU Time
  - Etc.
2. Allocating Stacks

- Allocate separate user-space stacks for each task in `kernel.c:_start()`
- In theory, processes have two stacks:
  1. User Stack: For the process to use
  2. Kernel Stack: For the kernel to use when executing system calls on behalf of the process

**Option:** In this assignment, you can opt to use only one stack
- Kernel threads need only one stack
- 4kB per stack is enough
3. System Calls - Typically...

- So user processes can ask for kernel services
- Standard Procedure:
  - Push system call ID + arguments onto stack
  - Interrupt / trap: elevate privileges + jumps into kernel
- NOT the case for this assignment...
3. System Calls - In this project

- User processes provided with `syslib.h`
- These functions:
  - Load kernel entry point address from known location in memory (`ENTRY_POINT`)
  - Push system call ID onto stack + call `kernel_entry` function
3. System Calls - kernel_entry

- `kernel_entry` address stored at `ENTRY_POINT` (0xf00)
- Saves registers + switches to kernel stack
- Does the reverse when exiting the kernel
4. Context Switch - Overview

- **Goal:** safely switch currently running task
- **When does this happen?**
  - Preemptive OS: typically when the OS tells you to
  - Non-preemptive OS: when task yields or exits
4. Context Switch - Responsibilities

1. Save task state into PCB
2. Push current PCB into ready or block queue
3. Choose new task from ready queue + pop its PCB
4. Restore new task state + run it
4. Context Switch - Saving State

● Tasks should not care what happens while it's not running - save current state in its PCB:
  ○ General purpose registers (including %esp)
  ○ Flags

● What about the instruction pointer?
4. Context Switch - Scheduling

- Kernel must maintain:
  - Ready Queue: tasks ready to be run
  - Blocked Queue: tasks blocked on some resource

- Which task runs next?
  - Regular: round-robin
  - EC: lowest run-time
5. Mutual Exclusion (via locks)

- Spinlock implementation is provided, you must implement a blocking lock
  - See spec for precise requirements
- No preemption => no race conditions *
- Exactly one correct trace
Timing context switches

- `util.c:get_timer` returns number of cycles since boot.
- Implement parts of `th3` and `process3`.
  - `process3` included twice in task list - be able to distinguish between the two executions.
Tips + Things to think about...

- What should you do when a kernel thread is run for the first time?
- What state should be saved to PCB? In what order?
- Get queue working in user space
- Code and test incrementally
(Monday, 10/08) Answer the questions:

- **Process Control Block**: What will be in your PCB and what will it be initialized to?
- **Context Switching**: How will you save and restore a task’s context? Should anything special be done for the first task?
- **Processes**: What, if any, are the differences between threads and processes and how they are handled?
- **Mutual Exclusion**: What’s your plan for implementing mutual exclusion?
- **Scheduling**: Look at the project web page for an execution example.
Questions?